Amendments to the Specification:

Please amend the paragraph beginning on page 5, line 10, as follow:

The AEP process adds automated error prevention practices (such as different breeds of automated testing, automated standard enforcement, automated monitoring, and the like) to prevent and expose errors throughout the entire software lifecycle. FIG. 1A illustrates how the AEP processes fit into the four main phases of a software lifecycle, "Design" 112, "Develop" 114, "Deploy" 116, and "Manage" 118. As shown in FIG. 1A, the rectangular boxes indicate the AEP practices (verification tools) that prevent errors. Similarly, the oblique boxes indicate the development practices corresponding to each phase of the life cycle that could introduce errors, such as, "code" 102, "integrate" 104, "deploy" 106, and "design" 108. For example, the AEP practices (verification tools) for "Enforcing Coding standards" 122, and "Unit Test" 124 (using the corresponding verification tools) fit into and correspond to the "Develop" 114 phase of the lifecycle. Similarly, the verification tools for "Load Test" 126," and "Integration Test" 128 relate to the "Deploy" 116 phase of the lifecycle. Likewise, the verification tools for "Monitor" 130, "Diagnose" 132, "Analyze Performance" 134, and "Analyze Functionality" 136 correspond to the "Manage" 118 phase of the lifecycle. AEP practices and technologies expose different types of mistakes and errors at the earliest possible phase of the software lifecycle. This early detection prevents errors from multiplying (as a result of code re-use and new code being built upon problematic code) and significantly reduces the cost of correcting the errors. Moreover, each time a problem is exposed, the manager or team leader uses the AEP method and technologies to determine what development phase or practice permitted the problem, and to modify the process so that similar errors cannot be introduced. Finally, the manager or team leader implements measures that gauge how effectively the modification prevents errors as well as verify that the team is following the modified practice.

Please amend the paragraph beginning on page 6, line 27, as follow:

AEP implementation can begin at any stage of an organization's software lifecycle and extend through all subsequent phases and cycles. The organization begins implementing the

AEP practices relevant to their current lifecycle phase (as shown in FIG. 1A), and then uses these practices to gauge whether the team and software should proceed to the next lifecycle phase. When the next lifecycle phase begins, the organization implements the AEP practices related to that lifecycle phase, and then repeats the process again. AEP peacefully co-exists with existing development process components such as change management, data management, design, testing, and so forth. In fact, AEP leverages, connects, and extends these components to boost software reliability and streamline the full software lifecycle. Moreover, AEP extends and promotes International Standard Organization (ISO) and Capability Maturity Model (CMM) efforts for process improvement. AEP's automation removes the main obstacle that typically impedes process improvement efforts such as CMM and ISO— if required practices are not automated, the process improvement effort usually decays and fails to deliver the desired result. AEP addresses and automates many of the specific "key process areas" recommended for CMM—especially those associated with high-level assessment.

Please insert the following paragraphs (from Parasoft.com) on page 8, line 10:

For example, Parasoft Jtest™ is a complete Java developer's quality suite for code analysis, code review, automated unit and component testing, coverage analysis, and regression testing — on the desktop under leading IDEs and in batch processes. Jtest™ provides software development teams with a practical way to ensure that their Java code works as expected. Jtest™ tool allows creation of custom rules by modifying parameters, using a graphical design tool, or providing code that demonstrates a sample rule violation.

Moreover, Parasoft C++test™ is an integrated solution for automating a broad range of best practices proven to improve software development team productivity and software quality. C++test enables coding policy enforcement, static analysis, comprehensive code review, and unit and component testing to provide teams a practical way to ensure that their C and C++ code works as expected. C++test can be used both on the desktop under leading IDEs as well as in batch processes via command line interface for regression testing. C++test integrates with Parasoft's GRS reporting system, which provides interactive Web-based dashboards with drill-

down capability, allowing teams to track project status and trends based on C++test results and other key process metrics. For embedded and cross-platform development, C++test can be used in both host-based and target-based code analysis and test flows. An advanced interprocedural static analysis module of C++test, simulates feasible application execution paths—which may cross multiple functions and files—and determines whether these paths could trigger specific categories of runtime bugs. Defects detected include using uninitialized or invalid memory, null pointer dereferencing, array and buffer overflows, division by zero, memory and resource leaks, and various flavors of dead code. The ability to expose bugs without executing code is especially valuable for embedded code, where detailed runtime analysis for such errors is often not effective or possible. Parasoft C++test™ tool includes a graphical RuleWizard editor for creating custom coding rules and provides automated code review with a graphical user interface and progress tracking.

Additionally, Parasoft® .TEST™ is an integrated solution for automating a broad range of best practices proven to increase software development team productivity and software quality. Parasoft .TEST™ ensures developers that their .NET code works as expected by enabling coding policy enforcement, static analysis, and unit testing. Parasoft .TEST™ also saves development teams time by providing a streamlined manual code review process. Parasoft .TEST can be used both on the desktop as a Microsoft Visual Studio™ plugin and in batch processes via command line interface for regression testing. Parasoft® .TEST™ includes a graphical RuleWizard editor (user interface) for creating custom coding rules Therefore, each of the verification tools is an independent tool with its own user interface, which can be executed in a batch process.

Please amend the paragraph beginning on page 14, line 6, as follow:

Data from testing and monitoring tools 38 (for example, Jtest™, C++Test™, WebKing™, SOAPtest™, CodeWizard™, DataRecon™, SOAPbox™, and WebBox™, from Parasoft ® Corp.) is retrieved by the data collector 32 and stored in the relational database 31. Access to the database is through a set of standard reports targeted for the various roles (e.g.,

architect, project manager, developer, and so on) within the different AEP solutions that GRS supports (Java Solutions, Web Development Solutions, and so on). Each of the above-mentioned verification tools has one or more scope parameters ~~parametesr~~ in a configuration file shared by the software developers. Also, these verification tool are capable of automatically generating one or more test cases. This provides the capability of executing these verification tools by any of the developers on the entire code without having to manually generate any test cases. One or more scope parameters in the shared configuration file can be changed based on the determined objective criterion of the quality of the computer software. In other words, the verification scope of one or more of the verification tools can be customized based on the verification results from execution of the verification tools on the entire code. This provides a feedback loop that improves the quality of the verified code.

Please amend the paragraph beginning on page 16, line 17, as follow:

The results of the nightly build run is then stored in the GRS server 33. The results are then processed to generate an objective indication of the code quality. In one embodiment, this objective indication of the code quality is a confidence factor, as described below in more detail. The objective indication of the code quality is a quality rating of the entire code that takes into account the verification results of each of the verification tools, the number of test cases run, and the success or [[of]] failure of the test cases, with respective to each developer and the entire development team.